Mobile App Development

CSE-4078

Lab Plan 1

Objective:

- To install and configure VS Code.
- To install and configure the Emulators.
- To install and configure the Flutter.
- To explore the VS Code functionalities.

Course Learning Outcome (CLO):

- **CLO-1:** Perform the execution, debugging, testing, and profiling of mobile apps in modern IDEs.

Lab Tasks

1 – Install and using VS Code

VS Code is a lightweight editor with Flutter app execution and debug support.

- VS Code, latest stable version

Install the Flutter and Dart plugins

1. Start VS Code.
2. Invoke **View > Command Palette…**.
3. Type "install", and select **Extensions: Install Extensions**.
4. Type "flutter" in the extensions search field, select **Flutter** in the list, and click **Install**. This also installs the required Dart plugin.

Validate your setup with the Flutter Doctor

1. Invoke **View > Command Palette…**.
2. Type "doctor", and select the **Flutter: Run Flutter Doctor**.
3. Review the output in the **OUTPUT** pane for any issues. Make sure to select Flutter from the dropdown in the different Output Options.

Creating a new project

To create a new Flutter project from the Flutter starter app template:

1. Open the Command Palette (Ctrl+Shift+P (Cmd+Shift+P on macOS)).
2. Select the **Flutter: New Project** command and press Enter.
3. Select **Application** and press Enter.
4. Select a **Project location**.
5. Enter your desired **Project name**.

Opening a project from existing source code

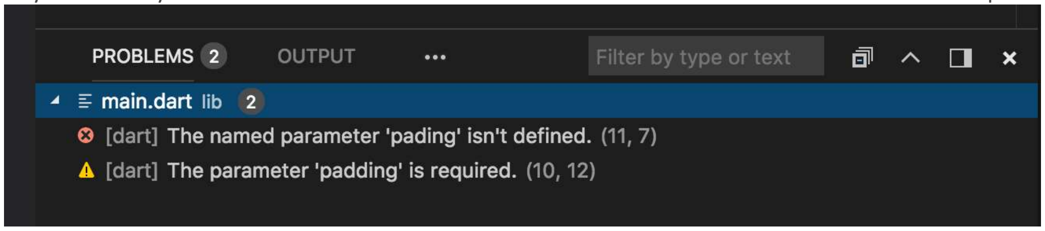
To open an existing Flutter project:

1. Click **File > Open** from the main IDE window.
2. Browse to the directory holding your existing Flutter source code files.
3. Click **Open**.

Editing code and viewing issues

The Flutter extension performs code analysis that enables the following:

- Syntax highlighting
- Code completions based on rich type analysis
- Navigating to type declarations (**Go to Definition** or F12), and finding type usages (**Find All References** or Shift+F12)
- Viewing all current source code problems (**View > Problems** or Ctrl+Shift+M (Cmd+Shift+M on macOS)) Any analysis issues are shown in the Problems pane:



```
[dart] The named parameter 'pading' isn't defined. (11, 7)
[dart] The parameter 'padding' is required. (10, 12)
```

Running and debugging

Note: You can debug your app in a couple of ways.

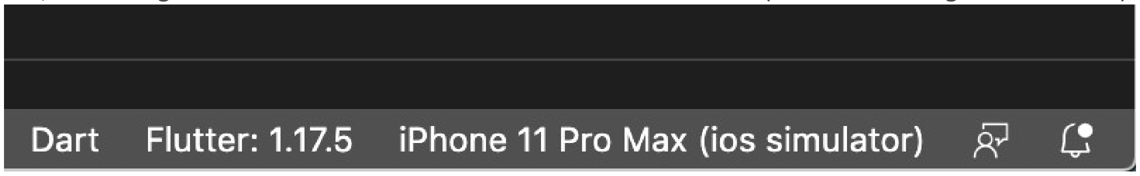
- Using DevTools, a suite of debugging and profiling tools that run in a browser. DevTools replaces the previous browser-based profiling tool, Observatory, and includes functionality previously only available to Android Studio and IntelliJ, such as the Flutter inspector.
- Using VS Code's built-in debugging features, such as setting breakpoints.

The instructions below describe features available in VS Code. For information on using launching DevTools, see Running DevTools from VS Code in the DevTools docs.

Start debugging by clicking **Run > Start Debugging** from the main IDE window, or press F5.

Selecting a target device

When a Flutter project is open in VS Code, you should see a set of Flutter specific entries in the status bar, including a Flutter SDK version and a device name (or the message **No Devices**):



Note:

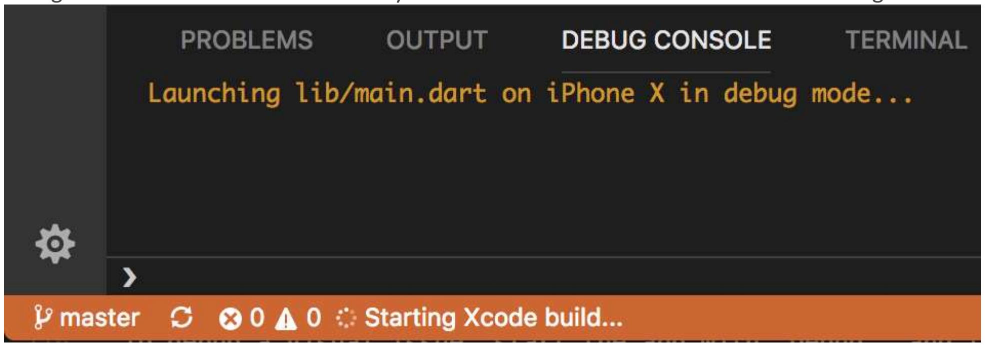
- If you do not see a Flutter version number or device info, your project might not have been detected as a Flutter project. Ensure that the folder that contains your pubspec.yaml is inside a VS Code **Workspace Folder**.
- If the status bar reads **No Devices**, Flutter has not been able to discover any connected iOS or Android devices or simulators. You need to connect a device, or start a simulator or emulator, to proceed.

The Flutter extension automatically selects the last device connected. However, if you have multiple devices/simulators connected, click **device** in the status bar to see a pick-list at the top of the screen. Select the device you want to use for running or debugging.

Are you developing for macOS or iOS remotely using Visual Studio Code Remote? If so, you might need to manually unlock the keychain. For more information, see this question on StackExchange.

Run app without breakpoints

1. Click **Run > Start Without Debugging** in the main IDE window, or press Ctrl+F5. The status bar turns orange to show you are in a debug session.



```
Launching lib/main.dart on iPhone X in debug mode...
```

Starting Xcode build...

Run app with breakpoints

1. If desired, set breakpoints in your source code.
2. Click **Run > Start Debugging** in the main IDE window, or press F5.
   - The left **Debug Sidebar** shows stack frames and variables.
   - The bottom **Debug Console** pane shows detailed logging output.
   - Debugging is based on a default launch configuration. To customize, click the cog at the top of the **Debug Sidebar** to create a launch.json file. You can then modify the values.

Run app in debug, profile, or release mode

Flutter offers many different build modes to run your app in. You can read more about them in Flutter's build modes.

1. Open the launch.json file in VS Code.

   If you do not have a launch.json file, go to the **Run** view in VS Code and click **create a launch.json file**.

2. In the configurations section, change the flutterMode property to the build mode you want to target.
   - For example, if you want to run in debug mode, your launch.json might look like this:

```
 "configurations": [
{
 "name": "Flutter",
 "request": "launch",
 "type": "dart",
 "flutterMode": "debug"
}
]
```

3. Run the app through the **Run** view.

Fast edit and refresh development cycle

Flutter offers a best-in-class developer cycle enabling you to see the effect of your changes almost instantly with the *Stateful Hot Reload* feature. See Using hot reload for details.

Advanced debugging

You might find the following advanced debugging tips useful:

Debugging visual layout issues

During a debug session, several additional debugging commands are added to the Command Palette and to the Flutter inspector. When space is limited, the icon is used as the visual version of the label.

Toggle Baseline Painting

Causes each RenderBox to paint a line at each of its baselines.

Toggle Repaint Rainbow

Shows rotating colors on layers when repainting.

Toggle Slow Animations

Slows down animations to enable visual inspection.

Toggle Debug Mode Banner

Hides the debug mode banner even when running a debug build.

Debugging external libraries

By default, debugging an external library is disabled in the Flutter extension. To enable:

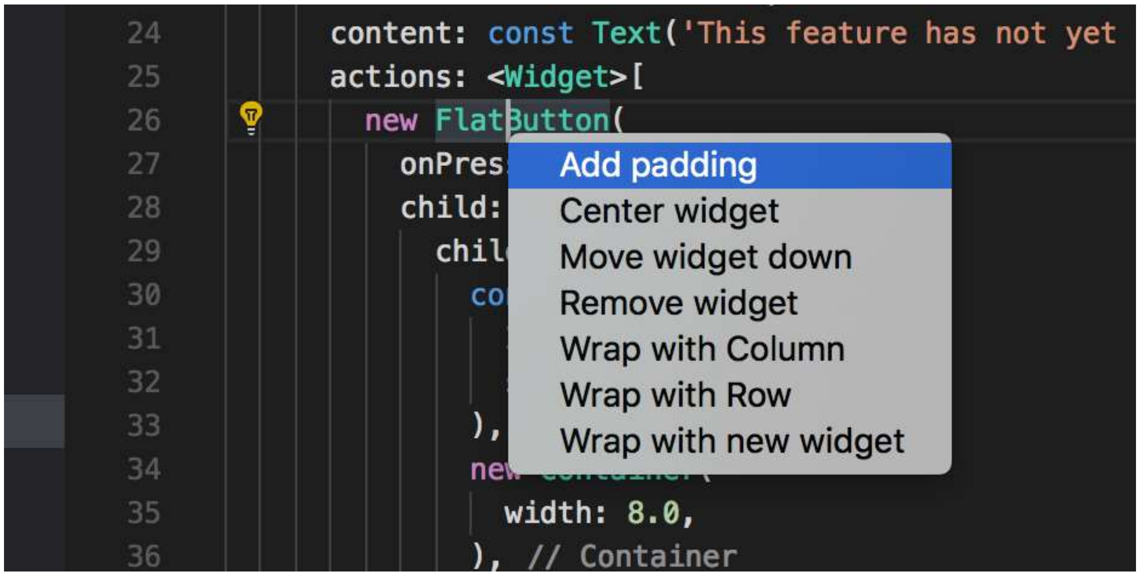
1. Select **Settings > Extensions > Dart Configuration**.
2. Check the Debug External Libraries option.

Editing tips for Flutter code

If you have additional tips we should share, let us know!

Assists & quick fixes

Assists are code changes related to a certain code identifier. A number of these are available when the cursor is placed on a Flutter widget identifier, as indicated by the yellow lightbulb icon. The assist can be invoked by clicking the lightbulb, or by using the keyboard shortcut Ctrl+. (Cmd+. on Mac), as illustrated here:



```
        content: const Text('This feature has not yet
        actions: <Widget>[
          new FlatButton(
            onPres
            child:
              chil
                co
                  
                  
                ),
                new
                  width: 8.0,
                ), // Container
```

Quick fixes are similar, only they are shown with a piece of code has an error and they can assist in correcting it.

Wrap with new widget assist

This can be used when you have a widget that you want to wrap in a surrounding widget, for example if you want to wrap a widget in a Row or Column.

Wrap widget list with new widget assist

Similar to the assist above, but for wrapping an existing list of widgets rather than an individual widget.

Convert child to children assist

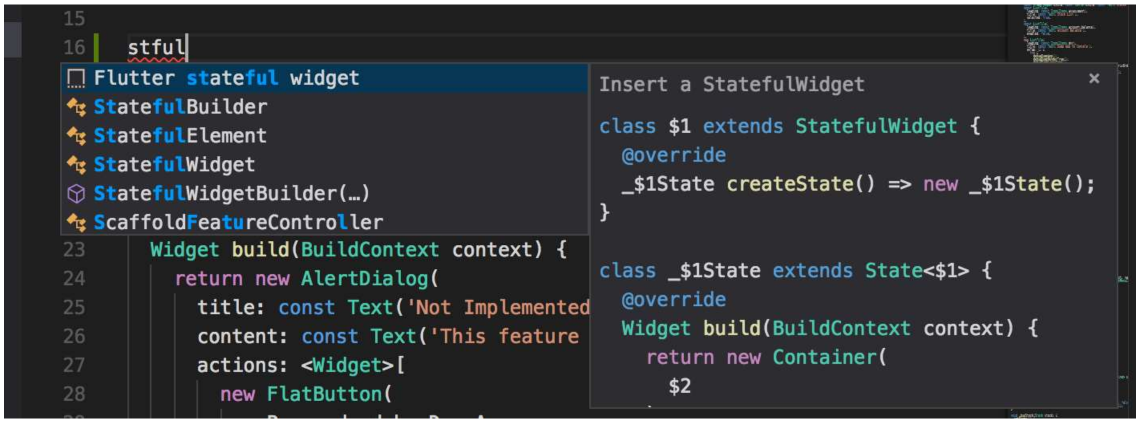
Changes a child argument to a children argument, and wraps the argument value in a list.

Convert StatelessWidget to StatefulWidget assist

Changes the implementation of a StatelessWidget to that of a StatefulWidget, by creating the State class and moving the code there.

Snippets

Snippets can be used to speed up entering typical code structures. They are invoked by typing their prefix, and then selecting from the code completion window:



```
  stful
  Widget build(BuildContext context) {
    return new AlertDialog(
      title: const Text('Not Implemented
      content: const Text('This feature
      actions: <Widget>[
        new FlatButton(
```

```
class $1 extends StatefulWidget {
  @override
  _$1State createState() => new _$1State();
}

class _$1State extends State<$1> {
  @override
  Widget build(BuildContext context) {
    return new Container(
      $2
```

The Flutter extension includes the following snippets:

- Prefix stless: Create a new subclass of StatelessWidget.
- Prefix stful: Create a new subclass of StatefulWidget and its associated State subclass.
- Prefix stanim: Create a new subclass of StatefulWidget, and its associated State subclass including a field initialized with an AnimationController.

You can also define custom snippets by executing **Configure User Snippets** from the Command Palette.

Keyboard shortcuts

Hot reload

During a debug session, clicking the **Hot Reload** button on the **Debug Toolbar**, or pressing Ctrl+F5 (Cmd+F5 on macOS) performs a hot reload.

Keyboard mappings can be changed by executing the **Open Keyboard Shortcuts** command from the Command Palette.

Hot reload vs. hot restart

Hot reload works by injecting updated source code files into the running Dart VM (Virtual Machine). This includes not only adding new classes, but also adding methods and fields to existing classes, and changing existing functions. A few types of code changes cannot be hot reloaded though:

- Global variable initializers
- Static field initializers
- The main() method of the app

For these changes, fully restart your application without having to end your debugging session. To perform a hot restart, run the **Flutter: Hot Restart** command from the Command Palette, or press Ctrl+Shift+F5(Cmd+Shift+F5 on macOS).

2 - Install and configure Flutter

System requirements

To install and run Flutter, your development environment must meet these minimum requirements:

- **Operating Systems**: Windows 7 SP1 or later (64-bit), x86-64 based.
- **Disk Space**: 1.64 GB (does not include disk space for IDE/tools).
- **Tools**: Flutter depends on these tools being available in your environment.
  - Windows PowerShell 5.0 or newer (this is pre-installed with Windows 10)
  - Git for Windows 2.x, with the **Use Git from the Windows Command Prompt** option.

    If Git for Windows is already installed, make sure you can run git commands from the command prompt or PowerShell.

Get the Flutter SDK

1. Download the following installation bundle to get the latest stable release of the Flutter SDK:

flutter_windows_2.5.2-stable.zip

For other release channels, and older builds, see the SDK releases page.

2. Extract the zip file and place the contained flutter in the desired installation location for the Flutter SDK (for example, C:\Users\<your-user-name>\Documents).

Warning: Do not install Flutter in a directory like C:\Program Files\ that requires elevated privileges.
If you don't want to install a fixed version of the installation bundle, you can skip steps 1 and 2. Instead, get the source code from the Flutter repo on GitHub, and change branches or tags as needed. For example:

```
C:\src>git clone https://github.com/flutter/flutter.git -b stable
```

You are now ready to run Flutter commands in the Flutter Console.

Update your path

If you wish to run Flutter commands in the regular Windows console, take these steps to add Flutter to the PATH environment variable:

- From the Start search bar, enter 'env' and select **Edit environment variables for your account**.
- Under **User variables** check if there is an entry called **Path**:
  - If the entry exists, append the full path to flutter\bin using ; as a separator from existing values.
  - If the entry doesn't exist, create a new user variable named Path with the full path to flutter\bin as its value.

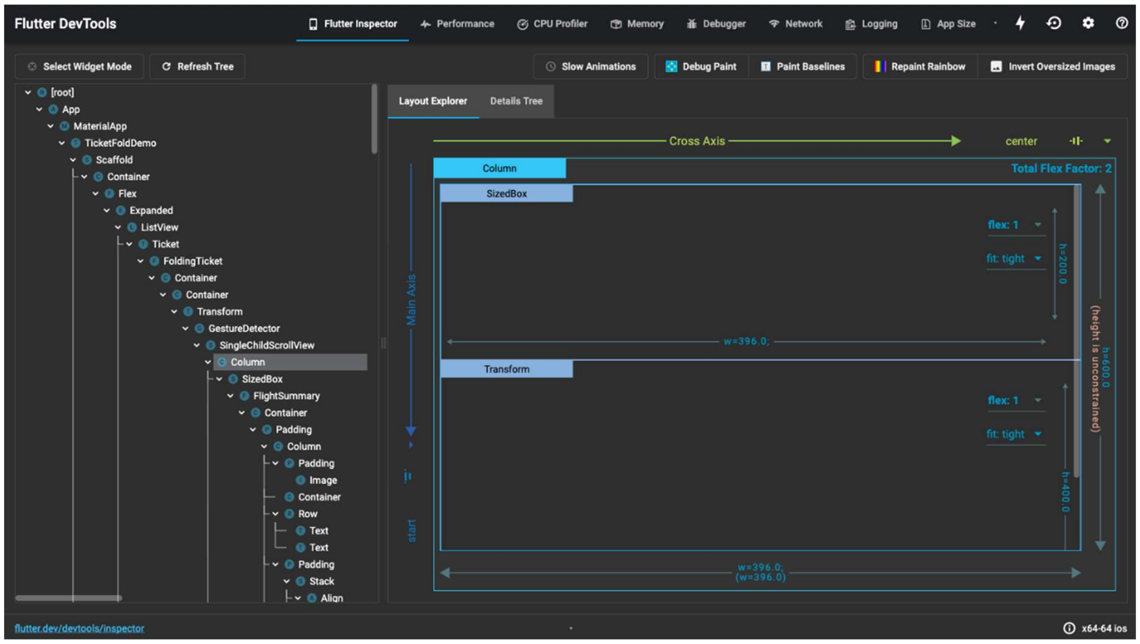
You have to close and reopen any existing console windows for these changes to take effect.

5 - Using the Flutter inspector

What is it?

The Flutter widget inspector is a powerful tool for visualizing and exploring Flutter widget trees. The Flutter framework uses widgets as the core building block for anything from controls (such as text, buttons, and toggles), to layout (such as cantering, padding, rows, and columns). The inspector helps you visualize and explore Flutter widget trees, and can be used for the following:

- understanding existing layouts
- diagnosing layout issues



Get started

To debug a layout issue, run the app in debug mode and open the inspector by clicking the **Flutter Inspector** tab on the DevTools toolbar.

Note: You can still access the Flutter inspector directly from Android Studio/IntelliJ, but you might prefer the more spacious view when running it from DevTools in a browser.

Debugging layout issues visually

The following is a guide to the features available in the inspector's toolbar. When space is limited, the icon is used as the visual version of the label.

Select widget mode

Enable this button in order to select a widget on the device to inspect it. For more information, see Inspecting a widget.

Refresh tree

Reload the current widget info.

Slow animations

Run animations 5 times slower to help fine-tune them.

Show guidelines

Overlay guidelines to assist with fixing layout issues.

Show baselines

Show baselines, which are used for aligning text. Can be useful for checking if text is aligned.

Highlight repaints

Show borders that change color when elements repaint. Useful for finding unnecessary repaints.

Highlight oversized images

Highlights images that are using too much memory by inverting colors and flipping them.

Inspecting a widget
You can browse the interactive widget tree to view nearby widgets and see their field values.

To locate individual UI elements in the widget tree, click the **Select Widget Mode** button in the toolbar. This puts the app on the device into a “widget select” mode. Click any widget in the app’s UI; this selects the widget on the app’s screen, and scrolls the widget tree to the corresponding node. Toggle the **Select Widget Mode** button again to exit widget select mode.

When debugging layout issues, the key fields to look at are the size and constraints fields. The constraints flow down the tree, and the sizes flow back up. For more information on how this works, see Understanding constraints.

Dart Questions:

1. Add two numbers and print the result.
2. Initialize two nullable variables.
3. Display the random number between 1 to 100.